Dynamic Hashing Algorithms

Albuquerque, New Mexico

Supercomputing Challenge

April 2nd, 2013

Team 49

La Cueva High School

Connor Brown Maxwell Sanchez Ryan De La O James Gavin Lewis

GPU-Hostile: Future-Proof Dynamic Algorithms; The benefits of implementing SHDA

Table of Contents:

Summary:

Introduction: Today, billions of people worldwide use the Internet for many purposes, from personal communication to finance management to mission-critical information exchange. Many such services work based on user-based access, with each user having access to certain resources and information. Account compromising can be a serious issue in many circumstances, and can result in fraud, theft, and more. Someone gaining access to an online banking account could mean hundreds of thousands if not millions of dollars in damage. The passwords used in user authentication are rarely stored in plaintext on the server though; they are instead hashed and stored in a database. When the user logs in tomorrow, the password he or she enters is hashed using the same algorithm as the first hash underwent, and the results are compared. If they are the same, he or she is granted access.

The Problem: The hashing algorithms that many companies use are cryptographically solid—Knowing the hash there isn't a known method for deriving the input text. For example, if I run SHA1 on the string "Computer", I get 924645b3e345a600f94ae78f01c5886cc320a89. There is no known way to reverse this base-16 hash back into the original password; "Computer". These algorithms, however, are extremely vulnerable to rainbow table attacks, where a hacker would have a precomputed table of all the possible letter-number combinations up to 10 characters in length, and when they run the above hash through this rainbow-table, it finds that "Computer" is the original password.

Just 15 years ago the thought of generating a rainbow table of any significance would be out of the question for any consumer or small business. Even a botnet would face a difficult dilemma of slow internet transfer speeds and storage for the portions of the generated tables. Today, a $400 GPU can compute over one billion SHA256 hashes per second, and this huge rainbow table can easily be stored on $500 worth of high-density magnetic storage. After a database dump, a hacker can start obtaining plaintext passwords immediately.

Currently, database managers and programmers have two main defenses against rainbow table attacks: salts and peppers, which alter the hashing algorithm or its output in some way. These work as additional data fed into the hash at runtime (concatenation), or a modification that occurs to the output post-computation. For example, a website may concatenate “6702” to every password entered, before the hash occurs: userPassword = SHA256(passw0rd + 6702). With the above example, even if the hacker did not know the salt, they could attack it with a rainbow table they pre-generated. If they generated a rainbow table for 12 characters, then their rainbow table could crack passwords up to 8 characters long from the aforementioned database, as all 8-letter password combinations would at some point be found with “6702” added on to the end of them. For example, if a user entered the password “penc1l” into the website, penc1l6702’s hash would be stored in the database. In a 12-character alphanumeric rainbow table, penc1l6702 would be an entry, and would be reported as the plaintext password. After seeing a trend of all passwords looking like <password>6702, the hacker would find that the salt is 6702, and could either stop and run with all the 8-and-below-character passwords they cracked, or could generate a new rainbow table using the above salt. As well, since this algorithm is a static algorithm, it would be a large hassle to upgrade the database to a new, harder (more CPU cycles to compute) algorithm to keep up with increasing computation speeds.

What We Want: We want a hashing algorithm that precomputation attacks are virtually impossible against. We want to make it as hard as possible for hackers to create a rainbow table after the attack, and we don't want a simple database dump to reveal the salt, like shown above. After the intrusion, we want the company implementing our algorithm to have as much time as possible to alert users, freeze funds, and assess damage before the hacker can recover the plaintext passwords of users. We want the algorithm to be controlled fully by the sysadmin, and we want the algorithm to become harder to compute over time, to combat generation of rainbow tables.

Implementation: The implementation of **SHDA (Secure Hashing Dynamic Algorithm)** we currently have is a Java Object that can be instantiated with the primes and a steporder:

```
hasher = new SHDA(prime1, prime2, prime3, prime4, prime5, “steps”);
```

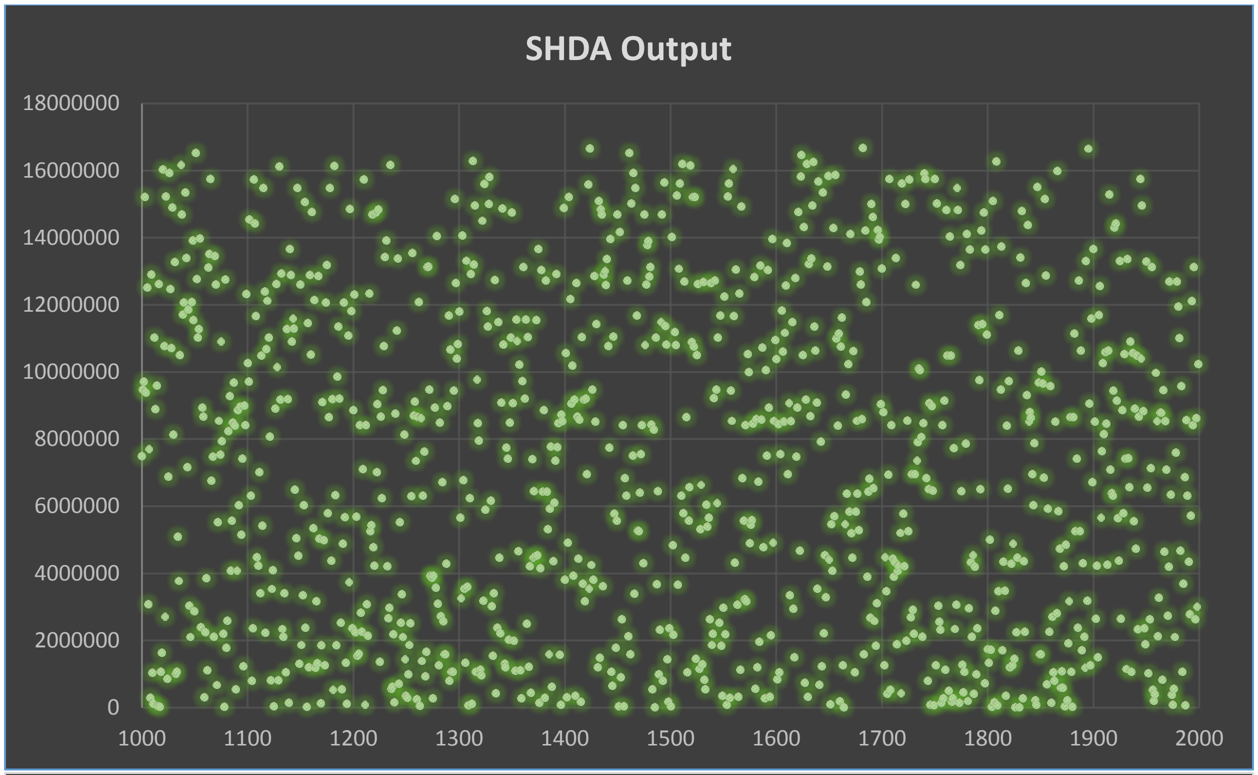
Where each prime is a prime represented in binary (with 32 characters and padding if necessary), and the step order is comprised of digits 0 to 9. Upconversion can be then done with this instantiated hash object, by providing the new step order plus the string to upconvert (as explained later):

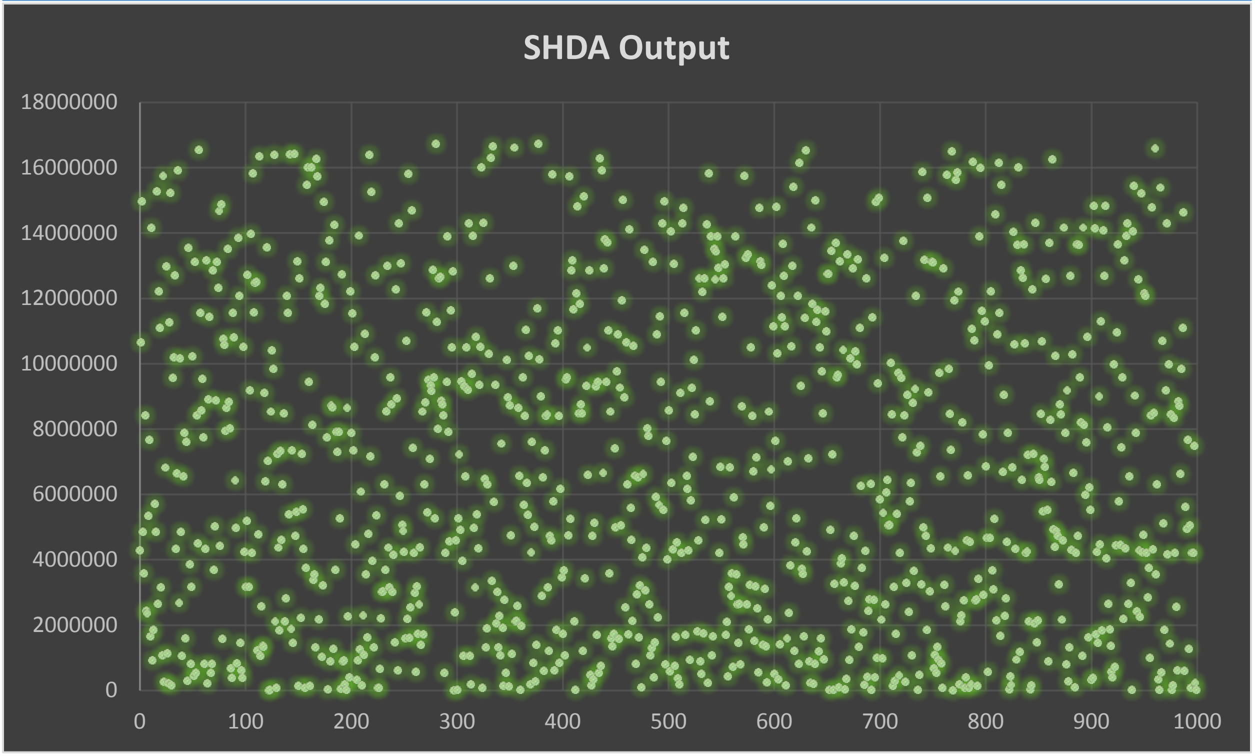
```
hasher.upconvert(password, newStepsToAdd);
```

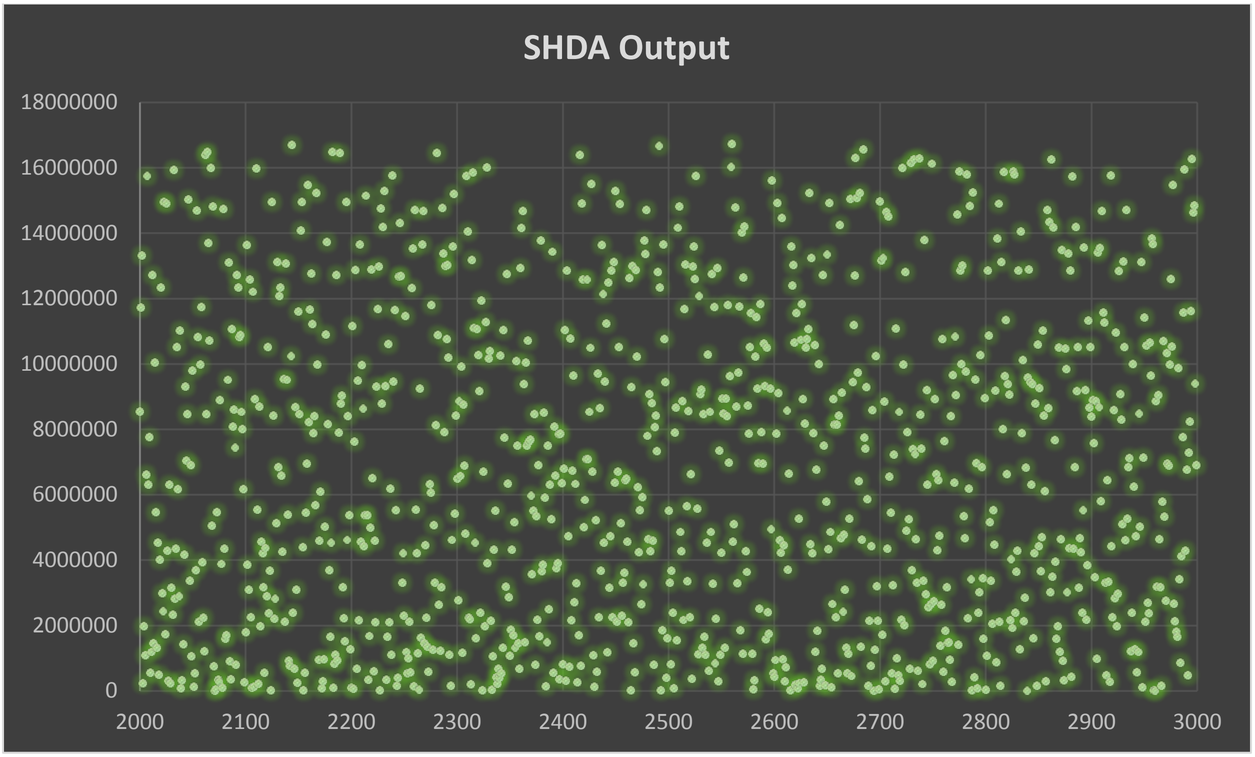
One of the primary difficulties in making a hashing algorithm is getting the results to have Gaussian (even) distribution. The charts below compare four different trials (each with a thousand inputs) with a sample area (as shown in the Java code below that generated it) converted from hex to base 10.

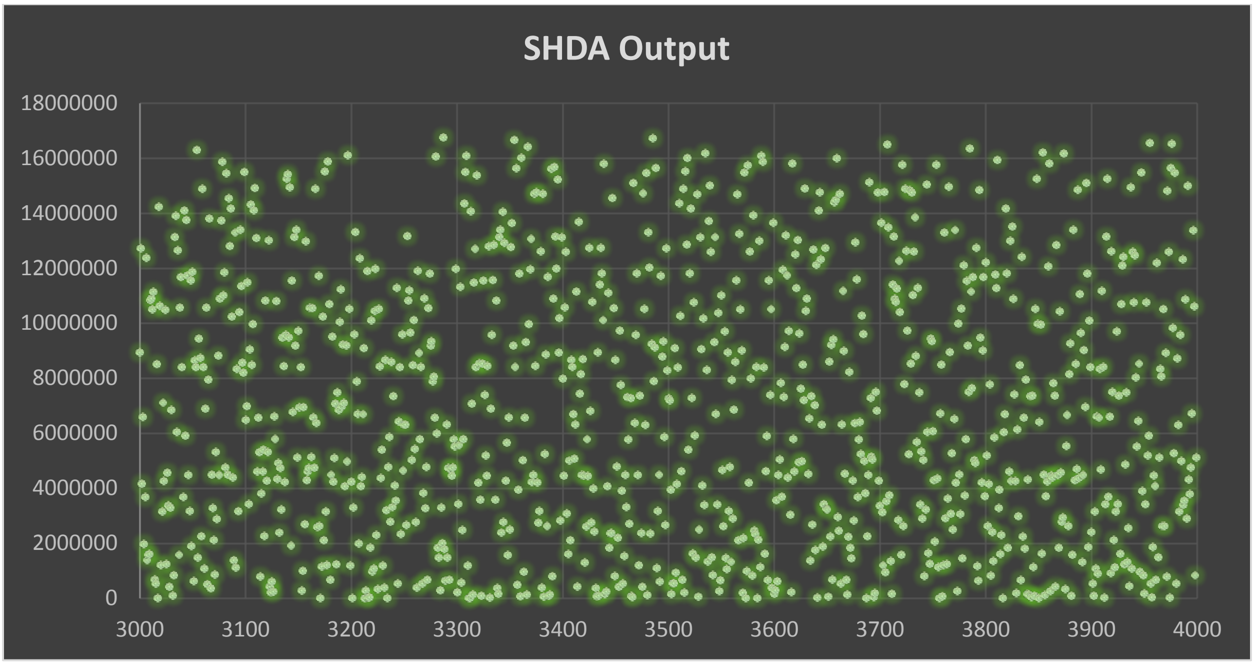
```
for (int i = 1000; i < 2000; i++) // or 2000-3000 or 3000-4000
{
    String temp3 = hash.getHash(Integer.toString(i));
    temp3 = temp3.substring(0, 20);
    Long l = Long.parseLong(temp3.substring(8, 14), 16);
    out.println(i+","+l);
}
```

The following graphs do not display any patterns vulnerable to exploitations that endanger the integrity of the output.

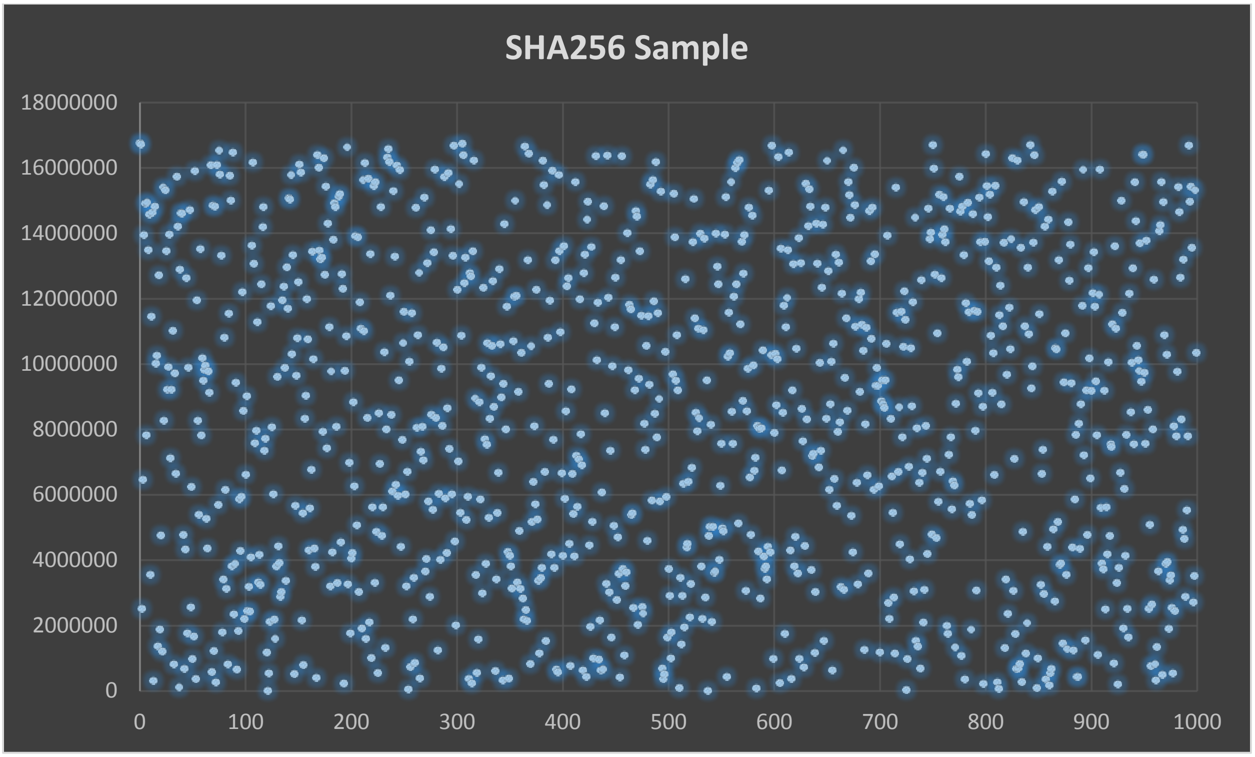

SHDA Output
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000

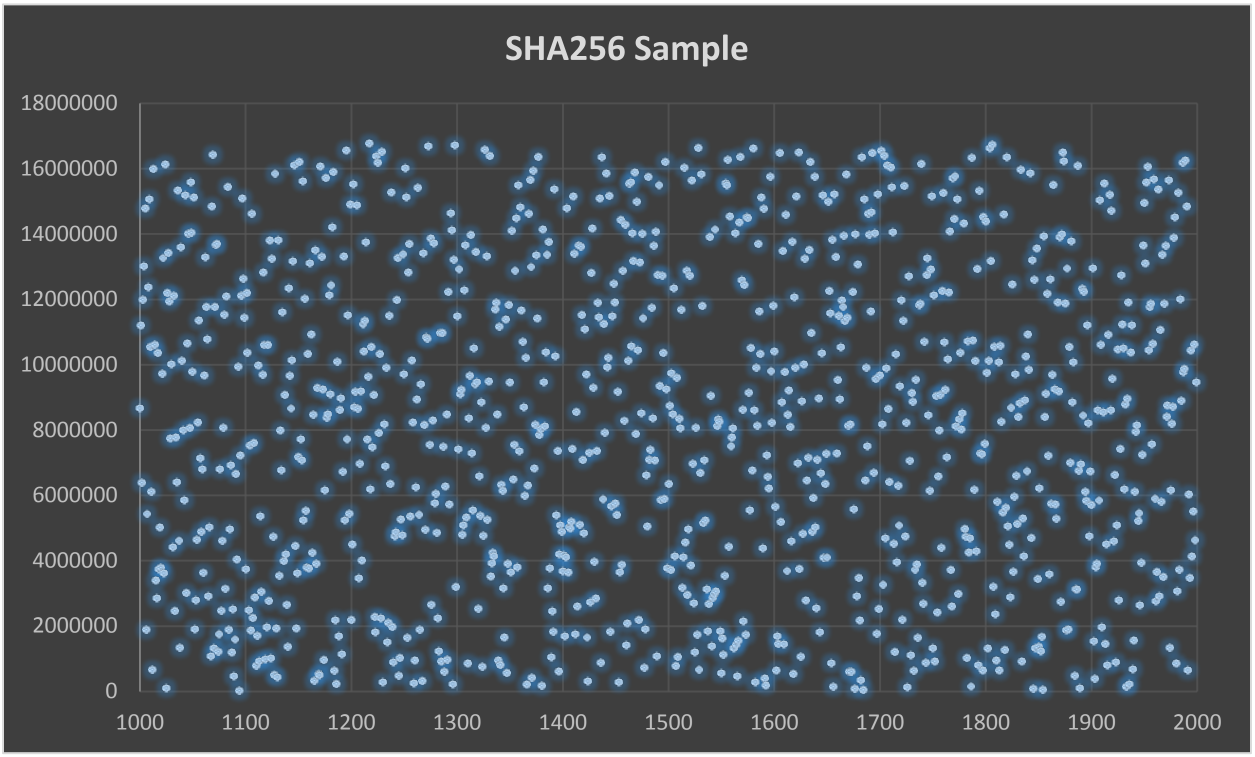

SHDA Output
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
0
100
200
300
400
500
600
700
800
900
1000

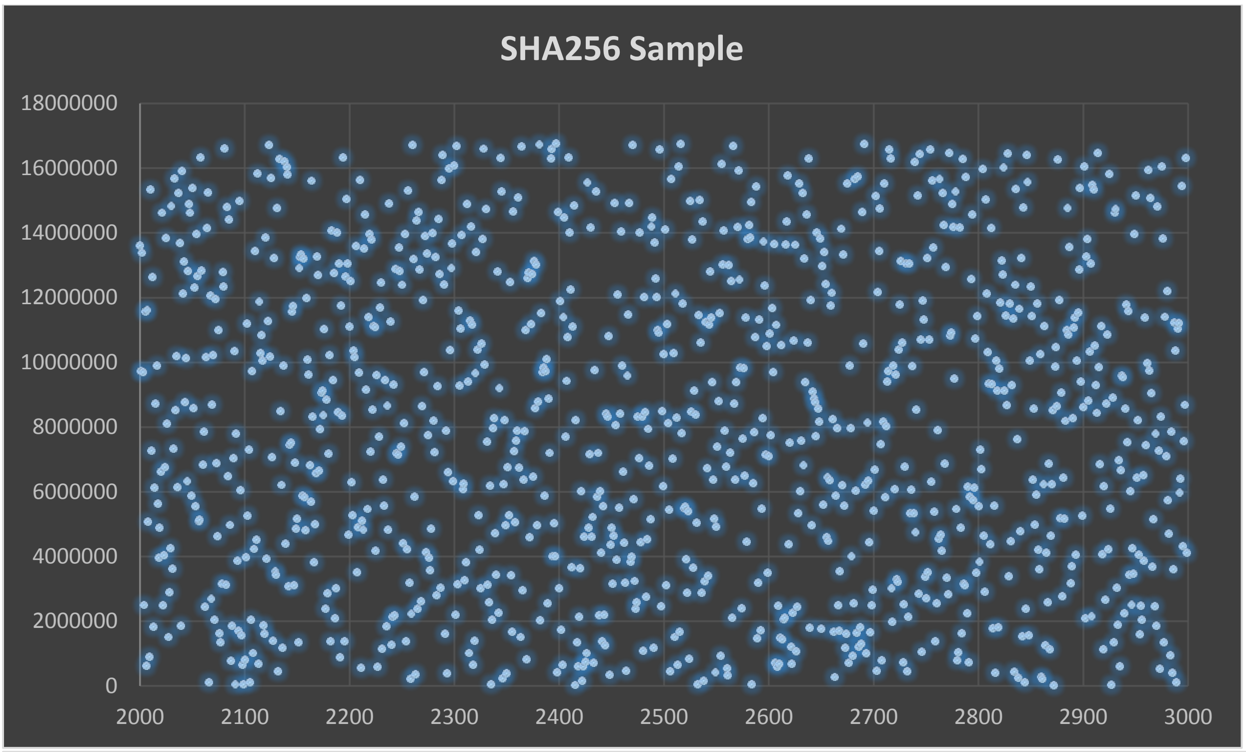

SHDA Output
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
2000
2100
2200
2300
2400
2500
2600
2700
2800
2900
3000

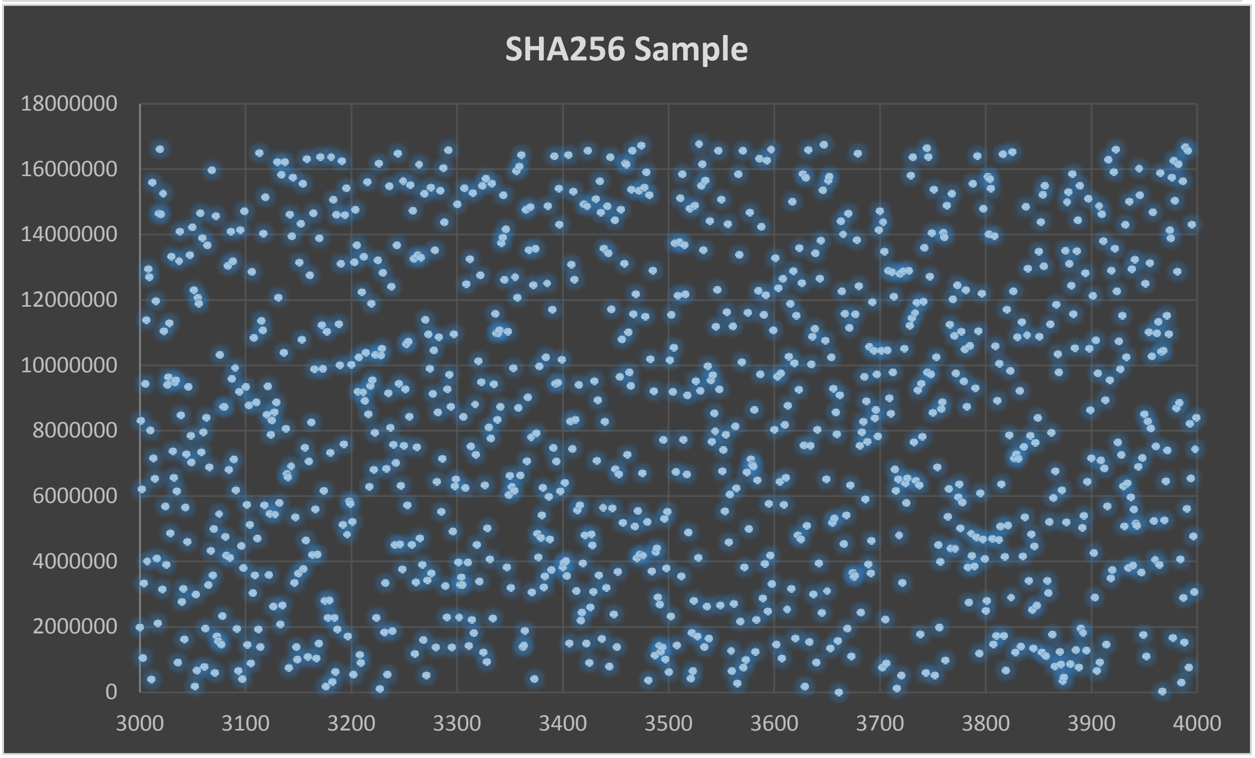

SHDA Output
18000000
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
3000
3100
3200
3300
3400
3500
3600
3700
3800
3900
4000

Here are some tables generated in a similar way using the popular SHA256 algorithm.









As you can see, our algorithm is close in distribution to SHA256, with a few more focus areas in our algorithm, likely due to many entropy-reducing steps.

Implementation of tri-level salting:

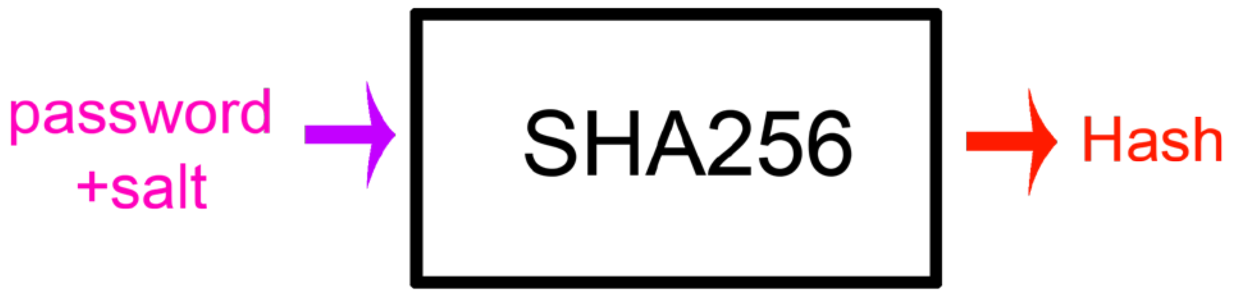
- **Classic concatenation/post-processing**
- **Starting prime numbers can be changed**
- **Number and order of algorithm steps can be modified**

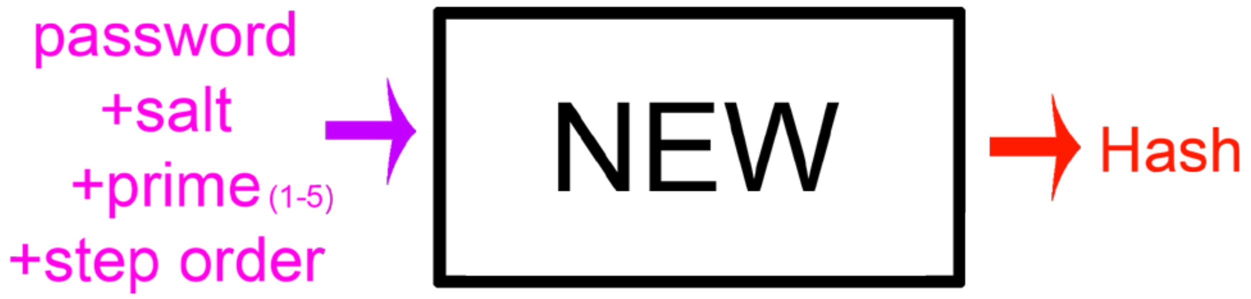
The concatenation/post-processing salt is implemented by the server administrator as normal salting is done with other algorithms. The starting prime numbers can be generated with very simple code:

```
Random random = new Random();

int prime1 = 0;

while (!isPrime(prime1) || (Integer.toBinaryString(prime1).length() != 31))

        {

            r2 = random.nextInt(1000000000)+1000000000;

        }
```

And then input the primes at object construction (new SHDA(prime, prime, prime, prime, prime);).

The step order is determined by the system administrator. Longer step orders take longer to compute, and are more resistant to rainbow table generation.





The salts embedded on the algorithm level help to resist Application-Specific Integrated Circuit creation. The high-memory steps slow down GPUs and Field-Programmable Gate Array devices, as accessing memory on these devices require time and leave cores idle.

To future-proof the algorithm, we have implemented a variable difficulty system, based on the step order. Since the steps are the last portion of the algorithm, additional steps can be added and hashes created using the old step order can have extra steps added onto the end of them.

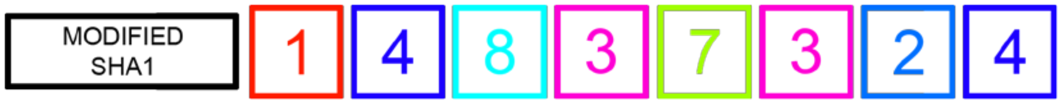
Steps:

With any portion of an algorithm that doesn't contain a one-to-one match, entropy (chaos, possible outputs) is lost. Some of our steps are made to give an exact one-to-one match (a form of mixing), which increase difficulty of computation while retaining all entropy provided to the step. Other algorithm steps do not provide one-to-one matching but are more secure at the expense of entropy. High entropy costs are only a problem when a large amount of steps are used, as eventually entropy boils down to a very limited output range. Mixing steps (not having more than two of the same step in a row) help to reduce the recurring patterns that some steps tend to produce in output. One of the steps is designed as a memory step, which when used makes the algorithm more difficult to run on high-performance consumer devices, such as Graphics Cards and FPGAs.

Here is the code for one of our entropy-reducing steps:

```
public static void one()
{
    String One = h0;
    String Two = h1;
    String Three = h2;
    String Four = h3;
    String Five = h4;
    One = leftx(One, 10);
    Two = leftx(Two, 12);
    Three = leftx(Three, 14);
    Four = leftx(Four, 16);
    Five = leftx(Five, 18);
    String[] str = new String[100];
    str[0] = One;
    str[1] = Two;
    str[2] = Three;
    str[3] = Four;
    str[4] = Five;
    int countOne = 0;
    int countTwo = 3;
    int countThree = 4;
    for (int i = 5; i < 100; i++)
    {
        str[i] = XOR(str[countOne], str[countTwo]);
        str[i] = AND(str[i], str[countTwo]);
        str[i] = OR(str[i], invert(str[countTwo]));
        countOne++;
        countThree++;
        countTwo++;
    }
    String temp = str[0];
    for (int i = 1; i < 100; i++)
    {
        temp = binaryAdd(temp, str[i]);
    }
    h0=leftx(h0, temp.length()%h0.length());
    h1=leftx(h1, temp.length()%h1.length());
    h2=leftx(h2, temp.length()%h2.length());
    h3=leftx(h3, temp.length()%h3.length());
    h4=leftx(h4, temp.length()%h4.length());
    h0=AND(h0, str[18]);
    h1=AND(h1, str[26]);
    h2=AND(h2, str[55]);
    h3=AND(h3, str[74]);
    h4=AND(h4, str[97]);
    h0=XOR(h0, str[19]);
    h1=XOR(h1, str[27]);
    h2=XOR(h2, str[56]);
    h3=XOR(h3, str[75]);
    h4=XOR(h4, str[98]);
    next();
}
```

To give a visualization of the step order:



In the case above, the step order is "14837324", with a difficulty of 8. If the sysadmin desires to increase the difficulty, he or she could add more steps after the 2nd "4", and the extra steps could be added to hashes that were created using the old difficulty 8 algorithm.

Application of our Work:

Our work would be mainly used by system administrators who have to manage user accounts in some form, and store/check passwords for those accounts. This algorithm could be used on most every website that uses logins with passwords. It is easy to implement for any website that uses java at some point, and future implementations of our algorithm will be in other languages such as the C family and common scripting languages.